02 BACKGROUND AND STATE OF THE ART

2.1 WebAssembly

2.1.2 WebAssembly ecosystems

The WebAssembly's ecosystem can be categorized into two groups: producers and consumers. The former ones, such as compilers, are in charge of generating WebAssembly binaries out of source code. **TODO** Do a rephrase here about the producers. what are the most common sources of Wasm etc

The latter ones, such as browsers, are in charge of compiling, analyzing and executing WebAssembly binaries. **TODO** Here, we discuss what happens after we get Wasm binary? What to do with it? **TODO** Mention, stress the landscape of tools that involve Wasm. Include analysis tools, fuzzers, optimizers and malware detectors.

TODO End up motivating the need of Software Diversification for: testing and reliability.

TODO The concept of managed and unmanaged data in Wasm from Lehman.

TODO Define: managed and unmanaged data, sections, execution stack, etc.

2.1.3 Binary format

WebAssembly's custom sections serve to save metadata, including details such as the compiler name responsible for generating the binary and symbol information pertinent to debugging. alters custom sections.

Wasm defines its own Instruction Set Architecture (ISA) [**?**]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types[1], each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

[1]`https://webassembly.github.io/spec/core/binary/modules.html#sections`

In Listing 3.3 and ?? we illustrate a C program and the Wasm program that results from its compilation. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in ?? shows the textual format for the generated Wasm. The module in this case first defines the signature of the functions (tpe1, tpe2 and tpe3) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (import1, export1 and export2). The definition of the function (func1) and its body follows the last import declaration at import1.

The function body is composed of local-variable declarations and typed instructions that are evaluated using a virtual stack (Line 7 to Line 32 in ??). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of ??, the result value of the main function is the calculation of the last instruction, `i32.add` at result. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. As the listing shows, instructions are annotated with a numeric type.

Wasm manages the memory in a restricted way. A Wasm module has a linear memory component that is accessed with `i32` pointers (integer of 32 bits) and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in data. The memory access is illustrated in load. This memory is usually bound in browser engines to 4Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in mem1 and export2). This ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (global1) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (call) or using a static table of function declarations. The latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm's compiler is in charge of populating the static table of functions.

In Wasm, all instructions are grouped into blocks, where the start of a function is the root block. Two consecutive block declarations can be appreciated in block1 and block2 of ??. Control flow structures jump between block boundaries and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (end1 and end2). By design, each block executes independently and cannot execute or refer to outer block codes. This is guaranteed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (break1 and break2) and table break. Each break instruction can only jump to one of its enclosing blocks. For example, in **??**, break1 forces the execution to jump to the end of the first block that starts at block1 if the value at the top of the stack is greater than zero.

■ 2.1.4 Types

■ 2.1.5 Memory model

TODO Managed and unmanaged

■ 2.1.6 Execution model

TODO Stack, frames and blocks **TODO** First order breaks
TODO Words on version 1.0 and why our tools have no issues with it.

■ 2.2 Software diversification

■ 2.2.2 Generating Software Diversification

■ 2.2.3 Variants generation

■ 2.2.4 Variants equivalence

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas.

■ 2.3 Exploiting Software Diversification

■ 2.3.2 Defensive Diversification

■ 2.3.3 Offensive Diversification